Object-Oriented Programming

12

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- History of object-oriented programming.
- Principals of object-oriented programming.
- Encapsulation and how it works in practice.
- Inheritance and how it works in practice.
- Polymorphism and how it works in practice.
- Abstraction and how it works in practice.
- Abstract class and how to use it in your program.
- Interface and how it is useful in designing inheritance.
- Future of object-oriented programming.
- Overloading and its use.
- Overriding and its use.

12.1 | Introduction

Programming languages allow computers to offer varied functionality. They enable us to create specialized applications that are capable of providing advanced functionality. There are different programming schemes that are used by different programming language platforms. This chapter focuses on the principles of objected-oriented programming, or OOP in short.

This chapter is divided into relevant sections. We will start by discussing basic OOP principles, which define the use of objects in programming languages such as Java. We will then discuss the use of classes and objects for implementing the same functionality, as produced by more direct languages such as C. We will discuss various subtopics that are essential for building a key understanding of the OOP world.

We will first discuss the basic principles of OOP languages and then study the other important themes that you must explore to learn more about the OOP world. Let us look at the history of programming principles, which shows how it was gradually possible to reach the OOP design of programming paradigm.

12.1.1 History of Object-Oriented Programming

The concepts of using objects and providing an orientation to computer code were developed in the early 1960s, especially by researchers at MIT. Programmers referred to code elements as objects, which have a set of properties. Simulate is the first language that presented the concepts of classes and objects, which are important for defining a language as one that employs OOP principles.

OOP languages improved data security and allowed programmers to produce data encapsulation in the form of creating private and public variables. Such compilers became popular and were employed to create programs for mainframe computers. The early OOP tools were especially efficient when used for carrying out complex tasks.

OOP principles especially got in popular use as they were perfect for creating programs that could follow human language-form instructions. The concepts of inheritance and encapsulation really caught on with various programmer communities, which understood that there were several benefits of switching from function-based programming especially for complex tasks.

The early languages started with smart ways to describe objects and create situations where it is possible to hide the implementation of the code to other programmers and program users. The programming environments gradually became available to smaller working situations.

There are two approaches that became widely used, with other approaches being dropped. Functional programming and OOP became the paradigms that were employed in all popular languages. There are many differences between functional

programming and OOP. One of the major differences is that functional programming follows stateless programming model and OOP follows stateful programming model.

C++ is an object-oriented language that was prepared from C, which is a fundamentally functional programming language. Software construction still followed instances where functional languages formed the basis of development while allowing programmers to use tools and libraries that implemented objectivity in their programming use.

OOP concepts gradually became popular as it was important to improve code security and provide control over data interfaces and class implementations. The OOP concepts gave rise to design patterns that are now commonly employed to resolve software problems in the modern development environment.

The use of behaviors and inheritance are the primary factors to incorporate an object-oriented design. This is a situation where it is possible to employ polymorphism and ensure the use of mutable objects. There are various methods that are still under the basic foundation of object-oriented principles. This may include abstraction, prototyping, and the use of singleton structures.

OOP languages have already surpassed the use of functional languages in modern use. These days, they are competing with the use of relational databases, as this ensures that it is possible to resolve all issues. According to computer scholars, the OOP paradigm is perfect when developers must create software systems that resemble human elements. It is perfect for taking a natural approach towards resolving problems by creating objects, which achieve the objectives required to resolve each scenario.

QUICK CHALLENGE Create a comparison chart to differentiate between procedural programming and object-oriented programming.

12.2 | Object-Oriented Programming Principles

OOP languages use the concept of creating every functionality with the use of distinct objects. This is different from the principle of creating functions that hold independent value. OOP principles call for creating data objects that provide a higher level of functionality and make it easier for the program developer to prepare a project according to the specific requirements of the client. Figure 12.1 shows the object-oriented principles.

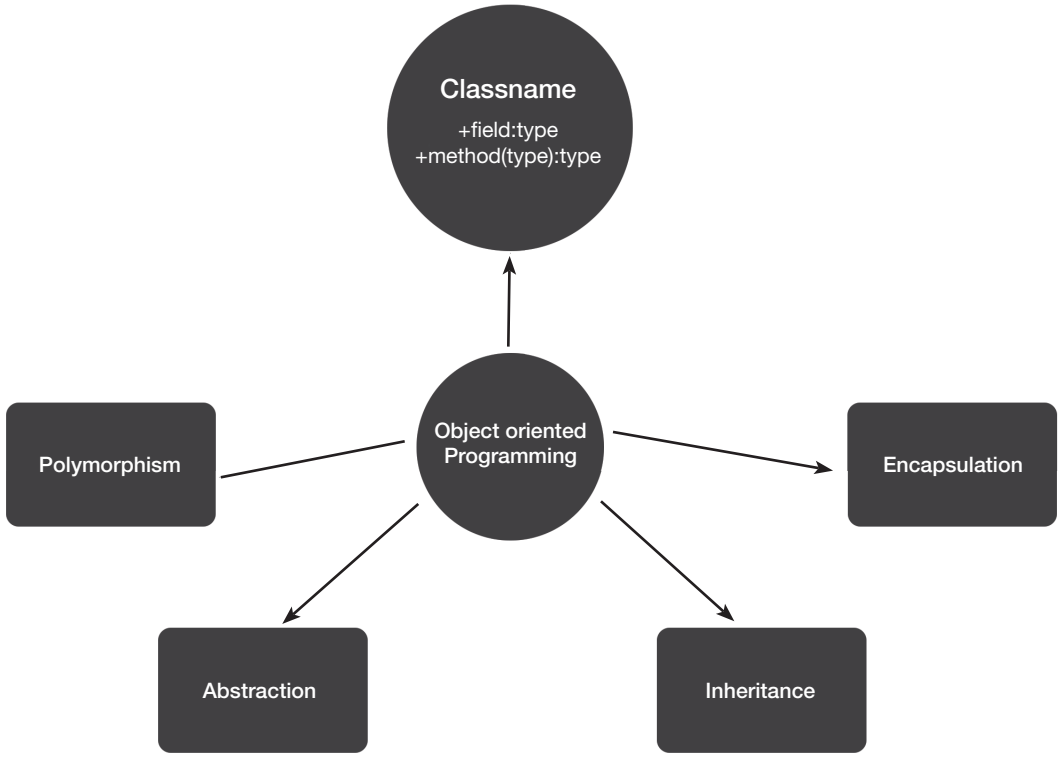


Figure 12.1 Object-oriented programming principles.

Regardless of any OOP language, there are four principles that define this technique.

1. **Encapsulation:** This is a method that separates data implementation from the user.
2. **Abstraction:** It describes the use of simple class objects that may provide the most complex of functions. This is a key principle of these languages and is often combined with encapsulation for an easier understanding and practical application.
3. **Inheritance:** It allows the creation of data hierarchy structures, which ensures that data objects can form related trees and branches. It creates the system of classes that are built within other classes to follow a systematic relationship for defining instances, variables, and implementing the functionality form all the upper level classes. In other words, every class that has super class gets access to variables and methods from all the super classes at every level.
4. **Polymorphism:** It is a more difficult concept to understand. In simple terms, it means that a single object may take on different forms, according to the defining principles of its use in the programming language.

Now, we will discuss these important concepts in greater detail. This will help you build a better understanding of the OOP concepts, and allow you to figure out how to go about using OOP languages for programming and application development.

12.2.1 Encapsulation

The first concept is the encapsulation of the data. Since data is arranged in the form of defined objects, they hold the properties of being distinct in their structure and is fully self-contained. The inner working of an object is defined by its state (attributes) and remains invisible to other parts of the code. The objects can show their behavior but maintain a strong boundary that separates them from other objects that are present in the programming environment.

Encapsulation works on the principle of hiding. The inner structure of all data objects is distinct. It contains all the elements, which are required to process it as a standalone part of the programming code. This objective is achieved by implementing boundaries that protect objects using specific tools. Access modifiers are used in a language such as Java, which allows you to hold full control of the attributes that define a data object. Figure 12.2 shows an example of encapsulation where all the ingredients are hidden inside the spring role. This can help you to visualize how encapsulation works.

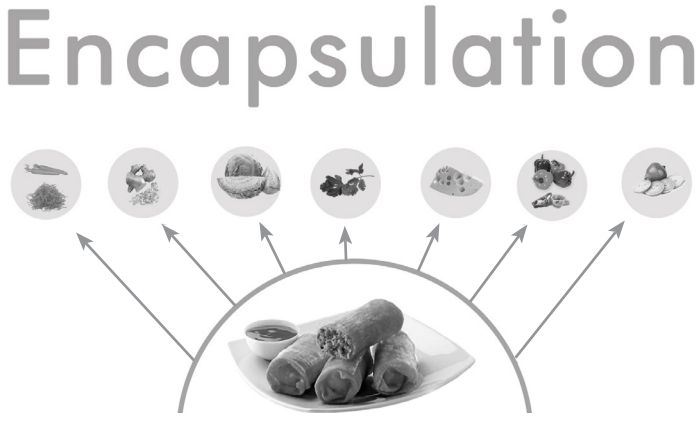


Figure 12.2 Encapsulation example.

There are some languages that create very strong encapsulation boundaries. Then there are languages like Java, which provide better control over the object structure by allowing programmers to set up different object property specifiers. However, encapsulation is still a primary feature in Java too.

This important principle allows us to separately keep data and the code sections that call for it. It allows programmers to change the original code, while never affecting the database objects that hold the legacy data, since they are always called but remain inaccessible for change due to a specific access structure.

Here is an example that shows how we implement the principle of encapsulation in Java by producing functionality, which remains locked with a boundary that separates the outside world from the class behaviors.

```
package java11.fundamentals.chapter12;
class DemoEncap {
    private int ssnValue;
        private int employeeAge;
        private String employeeName;
        // We will employ get and set methods to use the class objects
        public int getEmployeeSSN() {
            return ssnValue;
        }
        public String getEmployeeName() {
            return employeeName;
        }
        public int getEmployeeAge() {
            return employeeAge;
        }
        public void setEmployeeAge(int newValue) {
            employeeAge = newValue;
        }
        public void setEmployeeName(String newValue) {
            employeeName = newValue;
        }
        public void setEmployeeSSN(int newValue) {
            ssnValue = newValue;
        }
}
public class TestEncapsulation {
        public static void main(String args[]) {
            DemoEncap obj = new DemoEncap();
            obj.setEmployeeName("Mark");
            obj.setEmployeeAge(30);
            obj.setEmployeeSSN(12345);
            System.out.println("Employee Name is: " + obj.getEmployeeName());
            System.out.println("Employee SSN Code is: " + obj.getEmployeeSSN());
            System.out.println("Employee Age is: " + obj.getEmployeeAge());
    }
}
```

In this example, we have three private variables that are described during class initialization. They remain private and cannot be affected by the methods are used in a typical program. However, it is possible to use set and get methods to define these variables in a particular class, where they can be mentioned and used in the main method of the program. This will produce instances that create new data objects and hold value in them, while the actual initialization and the coding behind the variables remain separated in the class definition.

This program will print *Mark*, *30*, and *12345* from the object getting values, without ever affecting the definition of the variables or making it available for the program to alter them.

Encapsulation is slightly different from abstraction in the manner that it defines the combination of all concepts into a single item with the ability to hide its internal data, which is not directly accessible to a program user. Encapsulation creates low coupling where various code elements do not have to depend entirely on each other. This is an excellent programming practice, which efficiently uses the available resources and is achieved through this ideal OOP principle.

Encapsulation is also excellent in terms of allowing the data and functionality to remain available for a user, while still hiding the way it is implemented. There is no information about the way objects are supposed to work, while still understanding the data that it demands, and the methods that it contains to get the job done.

QUICK CHALLENGE

Based on the above-mentioned example of `DemoEncap`, write a program on a real-life problem to demonstrate encapsulation.

12.2.1.1 Advantages of Encapsulation

There are several advantages of encapsulation and they are the reasons for introducing languages that employ OOP principles. Here, we share the details of some important benefits:

1. It provides the advantage of creating flexible code. This is possible because we can implement a variable field in any way we want throughout the program. We can only use the different methods that remain within the classes that we have implemented. The class can be maintained directly, while various implementations occur as we see fit.
2. We can ensure that fields can be only either read or written. This is possible by avoiding the getter and setter methods. We can create a variable as a private one and then only use the get method. This will ensure that there will never be a change in the value of a particular field. Similarly, we will only use the set method to lock the variable value in only a written situation.
3. A program user will never know how the code works in the background for any variable present in an encapsulated class structure. This means that they only have access to employing get and set methods, where they want to bring value or set value to the variables. There is no way of finding out how the actual value would be assigned or read from the original variable, which remains private in a language like Java.

What are the disadvantages of encapsulation?

12.2.2 Abstraction

Abstraction is a concept which is best defined and described in line with encapsulation. It suggests that it is possible to develop classes and objects that are defined according to their functionality, rather than their programming implementation. This leaves us in terms of creating models for all our requirements.

These models serve as structures and never represent the presence of an actual item. The primary characteristics that define an object completely distinguish it from the other objects. Abstraction is also produced by creating conceptual boundaries for these defining characters, which ensures that all objects are understood by the viewers according to their specific perspective.

Abstraction means that we can create programming tools that we can employ multiple times. It defines an object without ever presenting the object for its instance. This can be understood by creating classes that effectively deal with related items such as recording the names and addresses of employees by using a class, which is designed for handling personal information.

Abstraction in Java can be best defined by using abstract classes. These are classes that you specify by adding the word "abstract" before mentioning the name of the class. This will lock the class and tell the compiler that it cannot be instantiated ahead. It is simply defining a class as an incomplete one. However, the incompletion can be present in any part of the class definition. This is again identified with the keyword "abstract" with the method that we want to leave undefined. Here is a simple example:

```
abstract class MusicInstruments {
   protected String nameofInstrument;
   abstract public void playInst();
}
```

On its own, this will be an abstract object that has a defined structure but actually points to particular code or functionality in a program. However, we use this abstract by creating an extension that produces a subclass:

```
abstract class NewInstrument extends MusicInstruments {
   protected int numberOfPieces;
}
```

Now, this is a creation of a subclass that defines some object of value which has an additional field, aside from the initial variable fields. We can now create a set of subclasses that can all add various values and can be used for an actual implementation in a

Java program. Java also has the capability of producing interfaces, which allow for the use of the same abstraction principle to carry out the intended functionality that remains hidden from the common users.

Flash Quiz Can a class extend multiple abstract classes?

12.2.2.1 Use of Abstract Classes

As we have seen above, Abstract classes set up abstract methods, which cannot have an implementation. You can have an abstract class which does not have any abstract methods. These classes lie between interface structures and the common class structures. You may again be thinking about the use of such classes, especially in relation to OOP programming.

They are created to work as the parent shell for derived classes, which will then contain the objects that will be actually processed during subsequent codes and calls in the program. This process is further shown by the following example:

```
package java11.fundamentals.chapter12;
public abstract class Animals {
    public void PrintInfo(){
        System.out.println(GetSound());
    }
    protected abstract String GetSound();
}
```

This shows that we have one defined method in the abstract class, while there is another abstract method of `GetSound()`. Our choice of not instantiating this method means that it can hold different forms based on the creation of descending classes, which may use it differently especially with the use of code overriding.

QUICK CHALLENGE Should you use interfaces over abstract classes? Give reasons for your answer.

12.2.2.2 Real-World Understanding

The principle of abstraction is best understood by understanding the principle of driving a car. Let us consider that it is a company car, where the company director is authorized to use it in any intended manner. He only needs to "call" the car driving function by contacting the relevant driver and does not require any specific details of the car. On the other hand, the driver is responsible for carrying out the functions required for performing the actual driving.

This means that there are two types of features. The first are the properties or the attributes. The director must know about them, such as the registration number of a car and the name of the driver tasked with driving it. The set of information may include other specifying details such as the color of the vehicle and its comfort level.

The second type of feature is the functions that the called car can perform. It can start from a particular office and take the company employees to a worksite. This may include driving, refueling the car, and other intended functions that are termed as methods. The controlling director, which works here as the "calling code" does not need to know about how they will be performed. He will simply order them to be executed to achieve the required functionality.

This example represents abstraction at its best, where we describe how OOP languages work for programmers, developers, and end users. OOP principles such as abstraction certainly makes it easier to simplify complex tasks and break them according to the required problem-solving steps.

12.2.2.3 Benefits of Abstraction

Abstraction offers several benefits and therefore, it forms the basis of all OOP languages. Here are some excellent advantages of using this specific principle:

1. It creates a barrier, which protects the implementation layer from the code users.

2. It also offers flexibility in terms of later changing the way implementation is carried out. This may be done to improve the coupling structure to loosen it up. A loose system is beneficial, as it allows all involved parties to make the best use of a working contract created through an application interface.
3. It makes it easier to perform debugging and find out which working layer is at fault for a particular problem.
4. It can be delivered through interfaces, allowing the easy correction of code use by an end user or the identification of wrong implementation scheme placed by the developer.
5. It also allows to set up for a divide and conquer policy for breaking a large program into smaller sections, which are easier to correct and implement by setting up interfaces that connect different parts of the complex program.

Flash Quiz Does abstraction restrict the functionality of the implementing class?

12.2.3 Inheritance

This is a principle that is specifically designed to improve the performance of structured programming languages. However, these languages produce a lot of duplicate code in a long program, since many code structures are often required in various places. The OOP principle of inheritance solves this problem by creating hierarchal sets of coding elements.

There are special classes that have the ability to copy the behaviors and attributes of their specialized functions. You can then only override the specific elements, which are required to be changed in different parts of the program. Your code becomes optimized, as each time the class object is called, a new specialization creates child classes and objects that only have the change in some of the elements.

The main class is termed as the parent class, which contains the source code. Each specialization results in a child class, each with its own set of altered parameters. This creates a situation where all copies will continue to progress in an independent manner. Take the example of classes that handle monetary values and strings as a package. Their children classes may include one that handles the salary information of employees, while one may store the data of the attendees at a function.

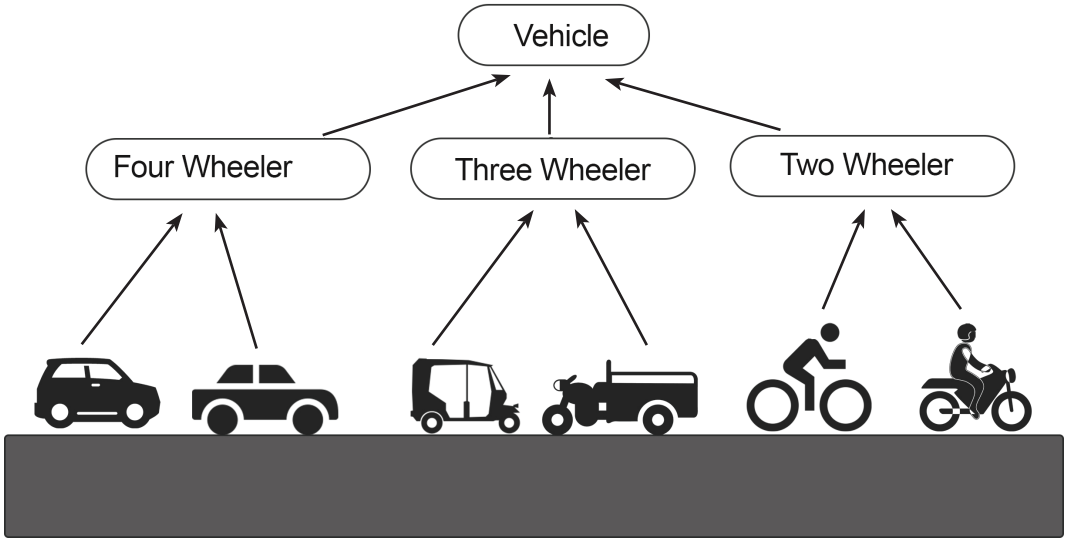


Figure 12.3 Example of Inheritance.

The following example gives a good idea of how inheritance is structured. In Figure 12.3, Vehicle is the superclass, which is also known as base class, and the others are subclasses of Vehicle. These subclasses inherit properties from the superclass Vehicle.

Remember, inheritance can be employed to set up a hierarchy of classes. Each addition only requires setting up new variables and data elements, while the basic set remains available to all the inherited classes. Inheritance simplifies the code and eliminates the instances where we may need to create the same data objects multiple times.

Now, let us see another example, where we create the basic class of a Tutor. We then create various extensions of the class to represent different tutors, like having a ScienceTutor and a LanguageTutor. These two extensions can be created as inherited classes, and only contain new variables that are important for a particular subclass extension. Here is a typical Java example for this situation:

```
package java11.fundamentals.chapter12;
class Tutor {
    String designate = "Tutor";
    String academyName = "NewAcademy";
    void performs() {
        System.out.println("Tutoring");
    }
}
public class ScienceTutor extends Tutor {
    String subject = "Science";
    public static void main(String args[]) {
        ScienceTutor obj = new ScienceTutor();
        System.out.println(obj.academyName);
        System.out.println(obj.designate);
        System.out.println(obj.subject);
        obj.performs();
    }
}
```

The output of the above program is shown below.

```
NewAcademy
Tutor
Science
Tutoring
```

This is an example where we first define the Tutor class that has two initial variables, which is the name designation and the academy name. Each inherited class can then add its own variables that will only be initiated for the members of that class and will not hold value for other objects of the Tutor class. However, we can block a subclass from using the members and methods of the original class if we define them as "private" during their initial declaration.

We can use the same technique to implement a set of well-defined classes that are present in a single hierarchy. This may not hold many levels when discussing people, but it is possible to have several subclasses when creating a complex Java program to imitate a complicated task.

It is possible to have different types of interference in Java. The primary inheritance is the single instance where there is a parent class, which gives rise to a child class. This is performed by extending a class to another class. This can be simply termed for two classes A and B, as B extends to A. However, it is also possible to implement multiple levels and create a large hierarchy, where each parent class extends a child and this occurs at least twice.

It is also possible to build a complete hierarchy, where multiple children classes are present and each has a single parent class. This is an excellent way of implementing functions, where we may have a single class structure that gives rise to objects which may also belong to other categories. Another type of inheritance is the seldom used multiple inheritance, where a single class may belong to multiple classes. Fortunately, this function is not available in Java 11.

Java employs the use of constructors when creating subclasses. It automatically creates inheritance, where all objects are constructed from the top to down method. It is possible to use a specific superclass constructor by employing "super" as the keyword. Remember, only one class can be termed as the superclass when creating a hierarchal setting class objects.

12.2.3.1 Advantages of Inheritance

There are several advantages of inheritance, which we have already described. Here, we summarize the primary benefits of the OOP principle of inheritance, especially as they are available when employed in the programming environment of Java.

1. It allows us to derive further classes. This creates a reusable model where we never change the existing classes, which improves the software development time required for program executions.
2. The derived classes generate an extended set of properties, which ensures that programmers create dominant objects that are fully capable of performing the required independent tasks with loose connections to each other.
3. Base classes can give rise to multiple derived classes, creating a dynamic hierarchy capable of providing excellent functionality under different conditions.
4. It is possible to create complex inheritance, which offers the benefits of having all the properties present in the base classes. (Not possible in Java.)
5. All common code belongs to the superclass and only needs to be executed and compiled once during a program operation.
6. It is possible to override methods, which is excellent for defining empty method definitions in base classes and then overriding them, according to the specific use cases.
7. It is possible to optimize the code and arrange the required functionality in an enhanced manner. This ensures that the program code is effective and provides better throughput results.

Flash ? Quiz Can a class have multiple inheritances?

12.2.4 Polymorphism

The last principle of object-oriented programming is harder to explain than the previous three principles. It provides the concept that it is possible to have objects that have the same hierarchal position, but behave differently on receiving the same message. Their behavior is different in terms of producing an output when intrigued by a calling code. Figure 12.4 gives an example of polymorphism by showing how one interface can be used by various classes so they can have their own implementations. Figure 12.5 shows how methods can be implemented differently.

Polymorphism works by altering the way objects behave to the calling code. This means that they will carry out their functions in a variety of ways, based on their particular nature. This is a concept that provides an advanced way of making use of objects that are present in a language such as Java. This complex topic is used in high-level programming, but we will make sure that we give some examples in this chapter that will help you understand this important concept better.

Figure 12.4 Example of an interface with different implementation classes.

Figure 12.5 Example of Polymorphism by showing different implementations of an interface.

A better understanding is possible when we define polymorphism as the capability of modifying some methods of an object through overriding. This means that particular objects will serve as subclass instruments of their parent class, when they are showing different behavior when employed using the same calling code and instructions. In other words, objects of the subclasses may have different behavior from the objects of the other subclasses derived from the same superclass. The methods of all the subclasses may return different results.

A good example in this regard is to think of a class that defines big cats. We know that there are various big cats out there, and each has modified methods that they use to stalk and catch prey. Now, the calling code is in the form of a command that asks a cat to catch their prey. Each cat (say, lion or jaguar) will use modified methods to carry out this task. In this manner, the same object works as belonging to different derived classes.

We can define two types of polymorphism, according to its implementation. These two types are overriding and overloading. In this section, we will get introduced to these concepts.

1. **Overriding:** Here, the compiler is responsible for selecting the method that will be executed after the call. The decision occurs each time the code is compiled. This is also termed as *run-time polymorphism*, because the object takes different shapes during the program execution.
2. **Overloading:** This is when the specific method that is used by the object is determined by the dynamic position and type arrangement of the object. This is defined not during the execution, but is fixed at the time of program compilation, which occurs prior to execution having fixed behavior. It is also termed as *compile-time polymorphism*.

Polymorphism certainly remains an important principle, and it is one that truly separates OOP languages from functional languages that have a fixed code behavior. Take the example of a class which has a method of animal sound. We implement it within a class of AnimalProperties. Now, we know that each animal has a distinct sound. We want to ensure that the method of `animalSound()` returns a different result that depends on which element we are processing.

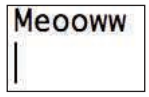
We will present two kinds of examples according to the two types of polymorphism that we have described in this section. Here is the first type of implementation:

```
package java11.fundamentals.chapter12;
public class AnimalProperties {
    public void sound() {
        System.out.println("This is top level animal class sound");
    }
}
```

Here are examples of how the same object behaves according to different object behaviors, using runtime polymorphism:

```
package java11.fundamentals.chapter12;
public class Cat extends AnimalProperties {
    @Override
    public void sound() {
        System.out.println("Meooww");
    }
    public static void main(String args[]) {
        AnimalProperties anmObj = new Cat();
        anmObj.sound();
    }
}
```

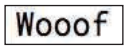
The above example produces the following result. Since we have overridden sound method, it will use the Cat class sound method to generate sound for Cat object.



Here is another way in which the same code would run differently:

```
package java11.fundamentals.chapter12;
public class Dog extends AnimalProperties {
    @Override
    public void sound() {
        System.out.println("Wooof");
    }
    public static void main(String args[]) {
        AnimalProperties anmObj = new Dog();
        anmObj.sound();
    }
}
```

The above program returns the following result.



The first runtime instance will print "Meooww" on the console, while the second one will present "Wooof". Each behavior is different, although the same class is employed but is designed to have methods that show different behavior according to the required object.

The second example for polymorphism employs the compile time method. This is a technique, where a programmer overloads the required methods to produce distinguished responses. The control for this polymorphism is produced by using the arguments that we are passing to our calling methods. Here is a typical example:

```
package java11.fundamentals.chapter12;
class OverloadingMethod {
    void printOutput(int a) {
        System.out.println("the first number is: " + a);
    }
    void printOutput(int a, int b) {
        System.out.println("The two integers are: " + a + " and " + b);
    }
    double printOutput(double a) {
        System.out.println("The double number is: " + a);
        return a * a;
    }
}
public class OverloadingDemo {
    public static void main(String args[]) {
        double results;
        OverloadingMethod omObj = new OverloadingMethod();
        omObj.printOutput(20);
        omObj.printOutput(20, 30);
        results = omObj.printOutput(2.5);
        System.out.println("The multiplication results is: " + results);
    }
}
```

The above program gives following output.

```
the first number is: 20
The two integers are: 20 and 30
The double number is: 2.5
The multiplication results is: 6.25
```

This is an excellent example where we use a single object and keep overloading it in the next instance. The first method of `printOutput()` that runs outputs a single parameter of integer type, which is 20 in our example. The next instance of the method `printOutput()` then takes two integers as arguments, which we set up as 20 and 30. The third instance is when the same object performs a simple multiplication of finding the square of the argument, which is 2.5. The resulting answer is 6.25, which is displayed on the console.

This type of polymorphism is termed as *static polymorphism*. This is because all the shapes that an object can take are already well-defined and take their place during the compilation. There are multiple definitions of the same method. A particular definition is selected based on the argument that we pass on to the method each time we make a call. This means that the method will become static, and will always follow our defined parameters.

This type is certainly great for producing better control over the direction of the methods that can have different instances. A good example is that of a simple calculator, which we only need to produce a fixed quantity of functions. It would be an ideal polymorphic presentation. However, there are certainly other situations as well; we must ensure that the program can behave dynamically to pick out the best course of action.

Dynamic polymorphism ensures that the problem of overriding is only carried out at the runtime stage of the program. This is excellent for controlling methods that may be implemented by using a hierarchy of classes, from our earlier explained principle of inheritance. Let us take the example where we have two classes. The parent class is termed as class First, and the child class is termed as class Second.

There is a method in the child class, which is designed to override the method – say, `exampleMethod()` – that belongs to the parent class. When we assign the child class object in such an example for reference, then this is used to determine which kind of object would be produced at runtime. This means that the type of created object will actually control which version of the same method is called in the Java program.

Remember, it is not important whether the object is now defined as the parent or in the child class. The method is selected based on the specification given to the new instance of the calling class, which will control the flow.

```
package java11.fundamentals.chapter12;
class First {
    public void exampleMethod() {
        System.out.println("This method is to be overridden");
    }
}
public class Second extends First {
    public void exampleMethod() {
        System.out.println("Now overriding the method");
    }
    public static void main(String args[]) {
        First obj = new Second();
        obj.exampleMethod();
    }
}
```

The above program produces the following output:

```
Now overriding the method
```

We have created an object that even though it belongs to the first class, uses the construction type of the second class. This will override the initial method and we will get the console out of "Now overriding the method" accordingly. We can mix and match these override situations, but the runtime execution will always select the method according to the type of object that it identifies from the constructor of the object.

QUICK CHALLENGE Think of any other real-life example of polymorphism and write a program for the same.

12.2.4.1 Advantages of Polymorphism

There are several benefits of the principle of polymorphism. It ensures that programmers can use smarter code design schemes and achieve the following advantages:

1. It ensures that programmers can first fully test and finalize their code before implementing it in a variety of ways. This way, it is possible to always use consistent elements in the final program.
2. Developers do not need to take care of the naming schemes, as polymorphism ensures that the same name can represent different data instances, such as int, double, and other available options in the OOP language.
3. It reduces the present coupling, ensuring that we can create flexible program objects. (We have discussed coupling earlier.)
4. It improves the code efficiency, when a program becomes long with complex functions placed within it, as it ensures that several instances and situations can be adequately handled.
5. Closely related operations become possible by using method overloading, where we can use the same name to designate different methods, each with their own set of parameters.
6. It allows the use of different constructors that can initialize class objects. This ensures that we have program flexibility, with access to multiple initializations.
7. It is carried out during inheritance principle application as well, as general definitions of a superclass can be employed by various objects that add their specific definitions, using the overriding of the available class methods that are not instantiated properly in the superclass.
8. It is excellent for reducing recompilation needs, by ensuring that even sections of a class can have a reusable structure, while the altering requirements may be achieved with the use of polymorphism.

Flash Quiz Are there any disadvantages of using polymorphism?

12.3 | Object-Oriented Programming Principles in Application

The OOP principles are important in programming since they offer several advantages. This method is perfect for applying on large programming projects. It ensures that we create reusable code and eliminate redundant elements in our programs. The combining of the running code with the data is perfect for creating self-contained modules that can then employ the advantages offered by modern computers.

OOP principles are excellent for several applications. The most important application is in creating dynamic applications that need updating every few weeks. This is often the case for security programs and ones that help in financial decisions. OPP programs are easily modified as they can have a modular structure. The chances of mistakenly entering wrong logic are also slim in OOP languages. While Java may not be a complete OOP language, it does offer the benefits that are associated with OOP principles.

Another primary application for OOP principles is in creating large-scale programs. These are complex programs that take on many inputs and then perform a number of calculations to produce the desired results. Creating objects allows programmers to set up distinct sections and produce privacy levels. All elements that must remain stable are defined in a private state, while ones that can benefit from an update are set up in a public setting.

Java and other OOP languages are great for creating server/client programs. These programs need efficiency and often require large memory and processing resources. These languages are excellent at creating real-time application solutions, which need to implement reactive programming elements. Parallel programming is best performed in OOP languages that can set up differentiated tasks and define a strategy that provides the ideal scenario for using concurrency during executions.

Artificial intelligence is another avenue for creating OOP programs. Here, we need to make automated improvements to ensure that machine learning becomes a possibility. Any programming application where efficiency and the ability to create smaller modules are required will be best served by using OOP principles and creating reactive programs. These are easily improved while ensuring that the primary legacy elements remain secure and invisible to programmers that may add further functionality.

It is also possible to have bugs in your large programs. OOP languages are excellent for performing the required debugging and the improvement of your code. The use of microbenchmarks can help you identify the performance of various program approaches, ensuring that you avoid logical errors and create a final program version, which is free from a buggy performance.

Since objects often follow inheritance and polymorphism, any errors are easily found. They can often occur due to the misuse of the objects and will seldom require the actual changing in the private object structure, which is another benefit of OOP. With these applications in mind, beginning programs should learn OOP principles and apply them in Java, which also offers primitives along with objects.

Can you use both interface and abstract on a class?

12.3.1 More About Objects

Objects are the primary unit of all OOP languages. They serve as elements that are individual and fully capable of communicating with each other. They can also form hierarchal structures and implement logical decisions based on the states and the returns of different operations. Objects are a collection of data types, elements, and their controlling and governing code.

Objects also follow the inheritance principle, where parent objects can give rise to various child objects. The child objects will always be the extensions and contain additional elements that provide a greater functionality over the basic object function and data structure. They are also excellent in terms of only holding the exact information required for their successful performance.

There are several benefits of using objects to employ OOP principles. They allow you to use programming in a way similar to the real-life decisions and understanding of concepts. We define each object by using specific attributes, while also defining a role for it. Programming languages mimic this behavior and provide the functionality of using logic and modularity for designing complex functions during software development.

It allows for realistic control designs that follow the ideal use examples. If we have a gear change system in a car, it always follows the available gears and cannot go beyond the available ratios that are provided by the components present in the transmission. We can implement the same behavior in Java objects, where they only behave in a predictable manner, by only following and taking on values that we have allowed in our object definitions.

Another advantage of using objects in OOP is the use of modularity. We can create independent objects which have the ability to provide a dedicated functionality in a variety of settings. The information that defines the objects is hidden from any instance of the calling codes. Therefore, it works perfectly for creating modules that can interact with each other but not produce any undesirable effects on each other during program executions.

Remember, Java objects work under the system of classes. It provides them an external structure. All Java libraries are a collection of classes, with several methods provide functionality by implementing reusable code, which has already been prepared by an expert for optimized Java functionality.

12.3.2 Classes and Object-Oriented Programming Principles

Classes are important in OOP languages like Java. This is because they allow programmers to use the same kind of logic that we use in the real world to define objects and classify them based on their properties and the function that they can perform.

Let us take the example of cars. There are different types of cars, but all of them can be defined as modes of transportation that use fuel to power locomotion. Your car can be simply defined as one of the instances of the class of cars. It may have certain properties and characteristics that are present in other cars. However, it will never show a property that makes it functionally different from another car. It can have additional behavior though, such as inclusion of a refrigerator and additional entertainment equipment. Figure 12.6 shows the difference between Classes and Objects.

The OOP principles are best acknowledged and used in programming when we employ blueprints. Classes work by providing an important structure, which then allows programmers to build up additional facility and produce improved programming results.

Class definitions only provide the structure and do not have execution element within them. This is provided by the `main()` method, which actually carries out the execution in any Java program. Once you set up these classes that define behavior which may be extended using additional values, it is possible to use the objects that embody the OOP principles that we have described above.

Classes allow the use of the principles of defining every functionality in the form of object instances. With the application of other additional methods, objects can then be used in a modular manner for the intended result. However, programming languages such as Java can go beyond the simple OOP applications and provide additional facilities.

QUICK CHALLENGE Create a food class and show all the possible child classes of that type. For example, fruits, vegetables, etc.

12.4 | Understanding an Interface

Objects need to interact with the environment around them in the programming application to ensure that they can perform the required interaction. They use methods that expose them and allow them to offer OOP functionality. Objects in Java use the interface to deliver their messages and interact with the running program code. An interface, therefore, is a set of methods which have empty bodies that certain objects can use for their particular function.

A language like Java uses the implementation of different interface elements to show a dedicated and definable behavior. The keyword of "implements" allows the creation of a class object which will implement the empty methods, ensuring that they are employed to allow the object to have a certain formal behavior.

The creation of an interface ensures that the program will only compile when the object uses all the methods that are present in that particular instance. However, this means that they are all successfully defined within that interface element for the successful compilation of the class during a program execution.

Interfaces further simplify the use of classes and objects in Java. An interface provides the answer to the functions of a particular code. Let us again consider the example of a car. A driver can use the steering to ensure that the car remains on the intended part. The steering wheel provides an interface here to use the transparent functionality.

However, the driver may not know at all how the steering process actually works, in terms of the drive axles, power steering mechanism, and the behavior of other mechanical components. The implementation in Java describes these details, which in most applications are redundant and are not required by the user. Interfaces are great to set up when a certain program must deliver a functional process, which is to be used by a normal software program user. This makes interfaces important, especially when employed with Java classes.

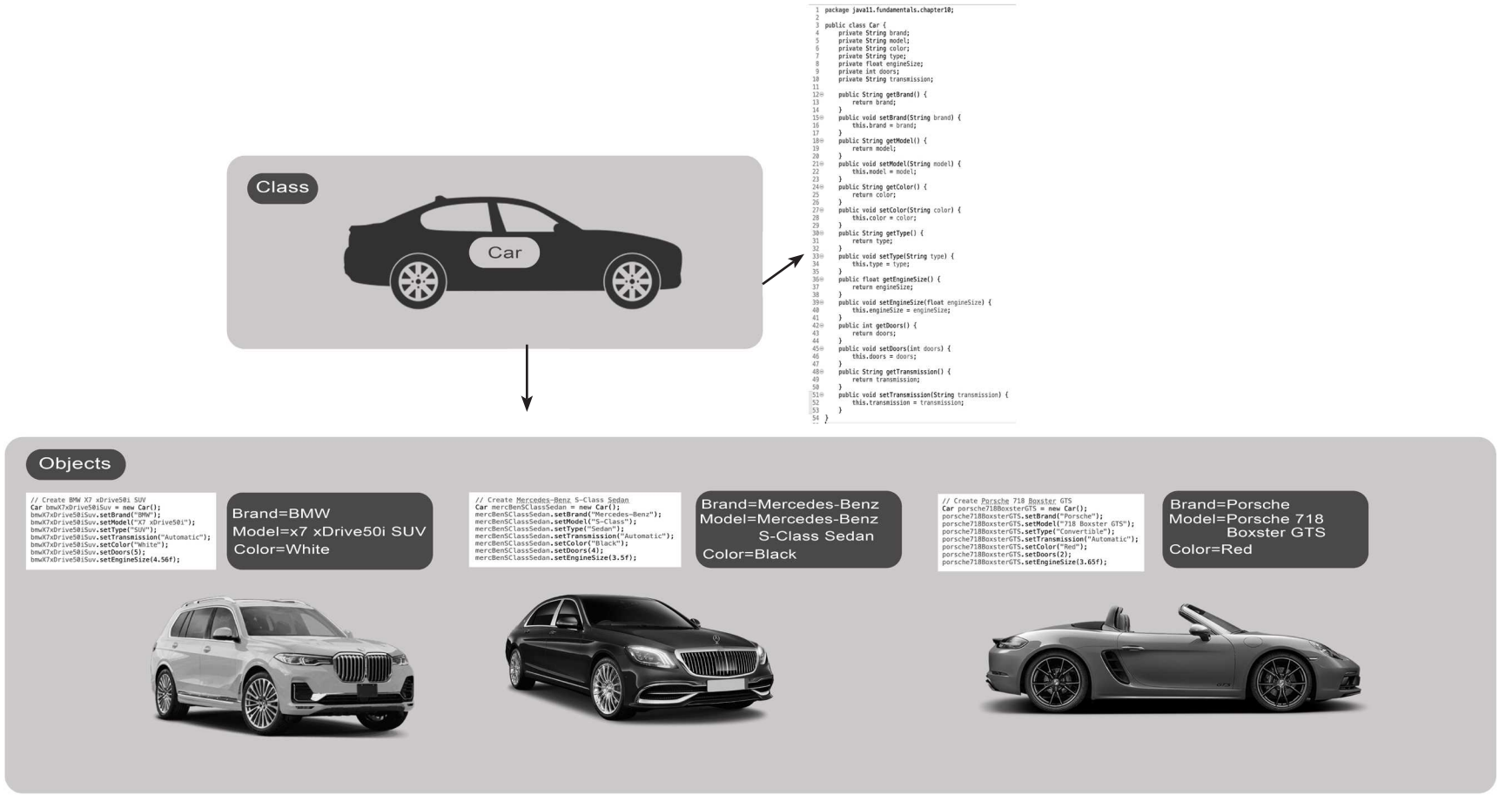


Figure 12.6 Example of Class and Objects.

Flash Quiz Can an interface extend multiple interfaces?

12.5 | Overriding and Overloading

There are two important concepts in Java known as overriding and overloading (Figure 12.7). We will explore both these concepts in this section.

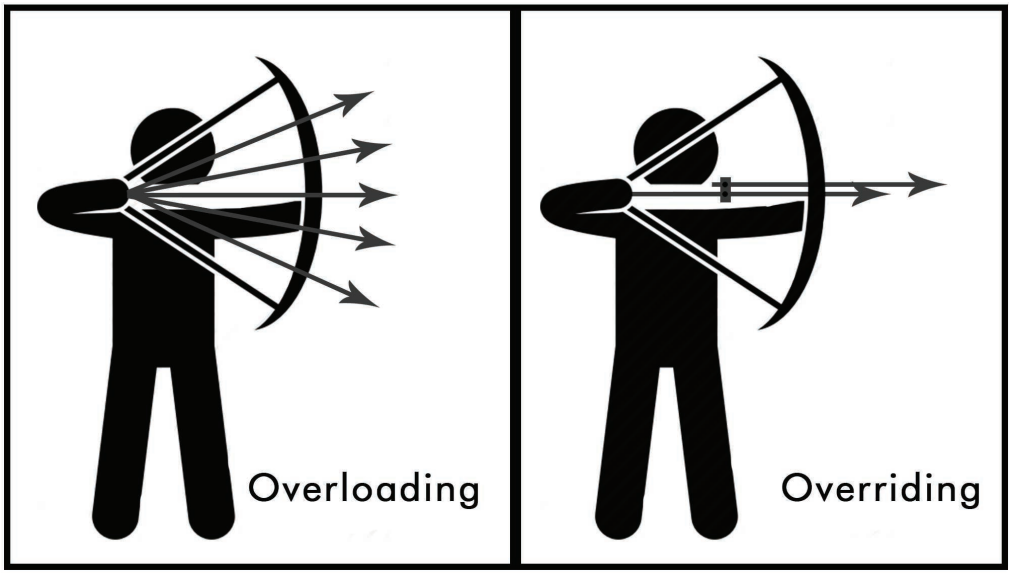


Figure 12.7 Explanation of Overloading and Overriding.

12.5.1 Overriding

Overriding is a process often used in Java programs. You will often create programs where you want to set up methods that you can alter when dealing with different objects or classes that are derived from a superclass. The simplest definition of overriding is that an instance method that occurs in the superclass will override the main method described in the superclass.

One way of using this technique is by writing "@override" annotation before the method. The use of this annotation produces an error from the compiler if it does not find the method first present in the superclass in a dynamic manner. In fact, the compiler actively finds that the method present in the superclass has been overridden in a subclass ahead. If it does not see an overridden implementation of a method definition, it will generate a compiling error.

Remember, overriding is often confused with hiding a static method. Overriding is different since it invokes the method which is described in the subclass. On the other hand, hidden methods can be invoked both from the superclass or the subclass.

In fact, overriding methods in Java offer an important example of an OOP principle. It allows the use of polymorphism. The overriding produces runtime polymorphism, where one object can have different conditions because of the different ways in which the methods can be implemented in the subclass.

Method overriding is easy to use by learning some basic principles.

1. The first rule is obvious, which is to write the exact name of the method as present in the superclass.
2. The second rule is to have the same number of parameters. Remember, a method changes how it behaves but this does not mean that the parameter arrangement can be altered.
3. The third rule for overriding method is the presence of a relationship. The methods can be overridden if they are present in a subclass, where they have an inheritance relationship in the form of the "Is A" model.

Here, we present a simple example of how method override will be employed in the syntax of a Java program. In the program, we present transportation as our superclass and describe the instance of a truck as the subclass, which invokes an overridden method.

```
package java11.fundamentals.chapter12;
public class Transport {
    void run() {
        System.out.println("We use transport vehicles for movement.");
    }
    public static void main(String[] args) {
        Truck obj = new Truck();
        obj.run();
    }
}
class Truck extends Transport {
    void run() {
        System.out.println("We use trucks for transporting loads.");
    }
}
```

This is a simple program, where the `run()` method has two different versions. When we run this method in the `main()`, we employ its second instance, when it will print the message of "We use trucks for transporting loads".

```
We use trucks for transporting loads.
```

This creates a condition where the same program is capable of running different responses based on the OOP principle of polymorphism.

Overriding is important when we have similar objects that we can place in a parent class, but the parameters or formulas that we want to employ in our program may differ. Take the example of banking calculations that will always use the formula, but there may be a change required in the interest rate to ensure that the calculation occurs according to the needs of the specific program object catering to a specific bank.

Remember, we override methods that are dynamic in nature. Dynamic methods are the ones that are called from the objects of a class, rather than the ones that are only described in the parent class. There is another method that is functional in other instances of polymorphism, which is called overloading.

12.5.2 Overloading

The other technique available for changing the methods present in a Java program and then using them for our specific function is the method of overloading. This is a feature where we can use the same name for multiple methods, where the method employed in the program depends on the arguments that we pass during a method's use. This is a technique where we can use multiple constructors that work on different argument conditions.

This syntax structure available in Java is just like using a constructor to develop the overloading function to have different class extensions. Overloading works by reading the details each time a method is employed in the main program. Take a simple example where we have a multiplication method. This method has two argument instances.

One instance describes the use of two numbers, while the second one is designed to calculate the multiplication of three numbers. Here are the two ways in which they can be run in the code, like having forms of `multi(int num1, int num2)` and `multi(int num1, int num2, int num3)`. We can clearly observe that the only difference between the two method calls is that their argument list is distinct from one another, apart from the actual variables that may be present in the calls.

This type of changing the method is the example of static polymorphism. This is because the type of object required is prepared at the compilation time, when the parameters from a method call are read to understand the required shape of a Java object. Here is an example that describes the use of overloading in Java syntax:

```
package java11.fundamentals.chapter12;
class OverloadMethods {
    public void displaying(char a) {
        System.out.println(a);
    }
    public void displaying(char a, int num) {
        System.out.println(a + " and " + num);
    }
}
public class OverloadExample {
    public static void main(String[] args) {
        OverloadMethods obj = new OverloadMethods();
        obj.displaying('A');
        obj.displaying('A', 100);
    }
}
```

The output of this program is in the form of two lines.

```
A
A and 100
```

This is a program which describes the syntax settings of the overload method functionality. We see two separate definitions of the `displaying()` method which both occur in the class initialization. We then create an object of our class, which then employs the variable two times. We only pass a single argument for the first time, which is A. This shows that we are looking for the method which only takes a single character datatype as an argument.

We then employ the method once again, but with two arguments. This automatically lets the compiler know that we are looking for overloading the method with the second definition, which takes as argument one character and one integer. The program automatically now calls this specific method definition and runs the program seamlessly. This is an ideal way of creating programs where you control the number of methods that you have.

You can tweak their definitions whenever you want to perform tasks which are quite similar, by simply presenting different ones in the class definition and separating them with the use of separate set of method arguments for specific use. The arrangement of the arguments, and the type of arguments are all important for providing the directions for the compiler to perform the method overloading. Each time the method definition, which exactly replicates the method call, will be employed in an active Java program.

However, there are scenarios where the Java language syntax is important for controlling the results of method overloading. The datatypes do not have to be fixed when calling the method. The concept of type promotion is employed. This is easily understood with the help of the following example.

If you have a method call with a floating number but the available method definitions only have them for a double integer, then the number will be changed into a double integer. However, this promotion only occurs when the compiler does not find an exact method definition for passing a call in the program code. The rules that govern type promotions in method overloading are:

1. byte to short to int to long
2. float to double
3. int to long to float to double
4. long to float to double
5. short to int to long

Always remember that since method overloading is controlled by having specific and independent set of definition arguments. The compiler will throw an error exception if it finds multiple method definitions that simply take similar arguments. Table 12.1 shows the difference between overloading and overriding methods.

Table 12.1 Overloading vs Overriding

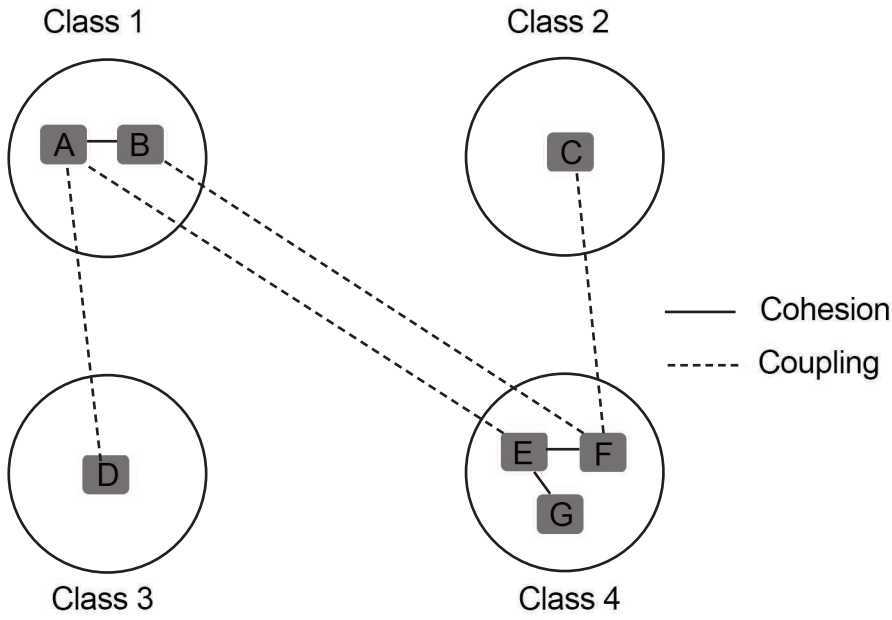
	Overloaded Method	Overridden Method
Argument(s)	Arguments must change in overloaded method	Arguments must not change
Return Type	Return type can change	Return cannot change. The only exception is covariant returns. A covariant return type can be replaced by a "narrower" type in case of overridden method in subclass.
Access	Access modifier can change	Access modifier can be less restrictive but must not more restrictive. For example, a superclass having a private method can be defined as protected or public in the subclass. But any public method in superclass cannot be declared as private.
Exception	Exception declaration in the subclass can change	Exception declaration can be removed or reduced but must not throw new or wider checked exceptions.
Invocation	At compile time, reference type determines which overloaded version should be selected.	At runtime, object type determines which method should be overridden.

Overloading and overriding are two very important points in OOP. You will be using these two often in your development career. Hence, it is important to master these two concepts.

QUICK CHALLENGE Think of at least five examples where you can apply overloading and overriding principles.

12.6 | Coupling and Cohesion

You will be able to deliver quality code and solve complex problems if you can embrace these two OOP design principles – coupling and cohesion. You should always aim for loose coupling and high cohesion. Let us take a look at what these two terms really mean.



12.6.1 Coupling

Coupling can be defined as the level in which one class has knowledge about another class. If they do not have any common shared knowledge; that is, if one class named A only knows class B via its exposed interface, these two classes can be termed as loosely coupled. On the other hand, a bad design could be if class A relies on parts of class B that are not part of class B's interface. This type of coupling is known as tight coupling, which you must avoid in your design.

12.6.2 Cohesion

Cohesion has to do with the level in which a class is self-contained. A single, well-focused purpose makes the class highly cohesive. There are two major benefits of high cohesive classes.

1. The classes that are highly cohesive are easier to maintain.
2. The well-focused nature of the classes makes them more reusable.

12.7 | Implementation in Java

Java is an excellent programming language and contains many Application Programming Interfaces (APIs), graphics facilities, and other options that are especially great at implementing the OOP functionality. It is a language which now offers the read-eval-print-loop (REPL) system, allowing for quick compiling of code to see how it will behave in a realistic environment.

The core Java features adequately cover all important OOP concepts, which include encapsulation, inheritance, and polymorphism.

Java also offers powerful factory methods that provide much more functional set of objects and classes. This comes with strong interfaces, which can describe any situation and ensure that it is possible to explicitly use code in a manner which ensures that no programmer will be able to use it in a wrong way.

Java is fundamentally a powerful language where you can use all the empowering principles of object-oriented paradigm. However, it still allows you to stay away from situations where you may feel limited by the creation of subclasses and the inheritance implementation structures. It usually imports interfaces and therefore, reduces the instances where we must implement classes in an unintended manner, which often creates further complexities in a Java program.

Creating the interface is important in most Java programs. Interfaces can reduce the number of methods and code that you need to carry out certain functionality. They are often employed with the @override annotation, which describes that the method in use is overriding its definitive function, whether it is described from a particular class or a specific interface. Remember, a failed compilation will never change a method in Java.

How many methods can a class have?

12.7.1 Objects Work as Solutions

Java is an excellent language because it allows the use of objects as solutions that resolve well-defined problems through the described parameters. The functions required from a program describe the problem, and then we end using multiple objects and interfaces to carry out the same functionality by using abstraction and employing inheritance at various levels.

Another way in which this situation works is by carrying out problem solving using the available states of objects. We set up the behavior of these attributes in a manner that the presence of the required data ultimately provides the situation, and where the ideal solution to the required solution is identified and achieved.

Most Java programs are created following a specific pattern. First, an industry expert works out the logical solution to a problem. This expert may know about programming or simply provide a human language algorithm or a set of required logical steps. Then, Java program developers use this available information to create an inheritance structure and set up the parameters required to deliver the intended functions through OOP principles.

Java offers multiple avenues for using objects. It allows setting up relationships which are ideal for eliminating code duplications while still ensuring the smart use of available programming avenues and advantages. The major advantage offered by languages such as Java when they first appeared on the horizon was to finally use a programming paradigm, where it was possible to eliminate the repeating code and therefore, create the fountain model of programming.

Object-oriented languages allow programmers to focus on the complex programs, while leaving the emphasis that was often placed at optimizing the code. Java is a language which uses the passing of messages for communicating with the different objects and creating a paradigm where the specifications of the message remain hidden from a common developer using them in a program, thereby creating an efficient working model.

This gives rise to the attributes and the behavior structure, which is shown by class objects in all programming languages. The attributes of an object is set up the way it is defined, especially when using the language to define solutions. The behavior depends on the arguments which are passed to different objects and the methods defined in a class. They employ the various modes of solutions based on the program inputs and outputs.

How many objects can you create in one class?

12.7.2 Ideal Tips for Implementing Basic OOP Concepts

There are several tips which you can follow when implementing OOP principles in your Java programs. Remember, the primary objective of object-oriented languages is to offer swift programming without jeopardizing the security of the prepared programs. The following are the main tips for implementing basic OOP concepts:

1. The first tip is to never repeat the same code in a language like Java. This is a key OOP objective, since we declare all tasks in the form of our required functionality. We can always call the related code accordingly by dividing our data objects in terms of the tasks that we need in a specific program. To avoid this, you should always create a method for a code section that you intend to use multiple times. This will allow you to simply call that method and process the data accordingly in each instance.
2. Encapsulation is the key if you want to continue to improve and alter your code in the future. This is a possible by setting up private methods and variables. You can always change this access to a protected one, which will allow you to always copy the code without ever affecting its primary behavior.
3. Your code should always follow the principle of singular responsibility. Each class that you define and use must cater to a singular need. This means that it will always be easier to call and extend your program classes without affecting the program by a significant margin. Avoid the coupling of key program functions. You can create class objects that combine results from different classes to perform the primary function, but avoid mixing multiple responsibilities in unique code units.
4. Another trick to follow is the use of the open-closed design. You must always keep your classes closed for any modification. This provides data security and ensures that you are following the OOP principles of abstraction and encapsulation. However, your individual methods and specific classes must remain open for further extension in terms of functionality.

   This is a great practice to ensure that you can hold on to your successful code, while still allowing your program to embrace new elements that result in improved functionality and scalable program behavior.

   You may be wondering how you can achieve the required functionality, if your program is complex and often needs to process large sections of calculations and data elements. The ideal trick to this is to use as many objects that you need. This is the main principle of using an OOP language to your advantage. You create specific instances, call the same methods from the class definition, get the particular task done, and then move on to run other objects in exactly the same manner.

   Remember, you do not have to create smart objects that perform complex functions. Use multiple ones and use them within each other for the intended functionality that should always have a singular nature. This allows programmers to truly employ the benefits of using the OOP principles to create an efficient code for any intended function.
5. Another tip is to judiciously use the interfaces that we have described above as special elements that help in OOP implementation. They are by design the helping tools for creating specific objects; use them in a contractual manner. Interfaces should be set up to give hints about the methods that should always be implemented in your source code.

   You can work in teams and divide the tasks that you must perform. This is possible if you use specific interfaces that force your employees or coworkers to ensure that they utilize the available methods properly. This creates a coherent code and ensures that all instances of a particular interface always allow the presentation of a unified set of methods.
6. Another ideal tip is the use of dependency injection. This is an excellent technique where the use of a static method or an object is employed for creating the dependencies and important needs of other objects. The dependency is also created using an inject, in which the injection is carried out using the transfer of properties using a client object.

This is an excellent functionality which allows Java developers to implement any type of testing. This technique forces programmers to create database objects which are easier to control using external functionality. This is perfect for testing and ensuring that you can fully obtain the benefits of programming in an OOP language.

7. Another key tip is to compose the inheritance of objects in the right manner. We can use these relationships and the concepts of objects being connected with each other to simply the OOP programming needs. When taken in the context of Java, this means that we should create classes which are simple and can always be extended. This is possible by the strong setting of inheritance and defining the functionality of each in a careful manner.

   Remember, an object may be a sub-element of a class, while still having a specific relationship with another object. A good example is that of an apple. An apple can be an object of the fruit class. It can then be described to have hard seeds. The distinction that we need here is that the main class must always be the fruit, as otherwise it will be difficult to define instances where a different set of characteristics is required.

8. Another tip is to create various classes that only connect to each other in a loose manner. If you create objects that are strongly attached to each other, it makes it extremely difficult to reuse your code to its full potential. Create situations where you perform inheritance, but keep the structure flexible enough to always implement additional functionality and the required attributes.

   You may have to first study and gather more information about the use of loose classes since the normal concept of inheritances to make a strong attachment between the objects. Remember, creating a code which is flexible and reused plenty of times is far better than creating a strong code which is perfect for use in a typical situation. Consequently, it becomes a burden when used elsewhere in the program because of its dependence on other objects for its optimal functions.

12.8 | Future of Object-Oriented Programming

A key element that we must discuss in this chapter is the future of OOP programming. As we move towards machine learning and the creation of smart code capable of carrying our self-improvements, it is important to consider the future of even the most successful paradigms of today. Here, we present arguments from both sides. We present the bright future ahead as well as describe some pitfalls that are on the horizon for the OOP world.

We first start by presenting a few downfalls. We believe that by knowing them, you can reduce the instances where you face these problems and learn to use OOP programming in the right way, where you become part of the future of OOP programming.

12.8.1 The Downfalls

Inheritance is the key for OOP programming languages. However, it may not be so easy to use in modern, complex programs where it is possible to create large hierarchal structures that will all need to be copied in your next program, if you aim to use a child class. This is especially difficult when your class objects also refer to other objects, because you will then require the class hierarchy from all involved objects.

Programmers describe this as the banana problem. If you want to copy a banana from a program code, you will end up copying the banana, its tree, its environment, and the whole jungle just to ensure code consistency through the class structure. This problem appears because of the implicit environment, and computer scientists are already looking at how to control and manage the problems that may appear because of extensive use of class hierarchy structures.

Another problem is produced by encapsulation, which often produces a complex situation during code compilation. A code reference is passed, where it may be required to create a complete and detailed clone object to produce the required level of functionality. Polymorphism is described as an OOP principle, but it can be easily implemented in other ways. In fact, Java allows you to use specific interfaces that produce the same functionality but in a more organized manner.

However, these are common issues and you can work around these problems. The future of the OOP world remains bright, as we describe in the following subsection.

12.8.2 Future Applications

There are various applications that are ripe for use in the future, and are already on the horizon. Take the example of web APIs, which are being prepared with the use of OOP languages such as Java. However, these functions are better prepared using functional programming languages, and current OOP languages also create a shell that produces the same functionality.

One of the best applications of OOP is in the field of user interfaces. User interface (UI) can produce situations which are uncalled for and cannot be resolved by having fixed structures. This is perfect for an OOP language because it is an excellent approach to build the dynamic structure. OOP languages are also great at producing frameworks, as there are several communities out there which provide support for platforms such as the Java Community that supports all Java functions.

Another application is in the field of cloud-based software solutions. Software as a Service (SaaS) is a particular application where OOP languages offer a winning proposition. OOP languages are perfect for creating solutions that can take advantage of the available resources, providing high processing functionality, and using optimum memory resources to produce high impact applications.

There is a strong shift to the cloud and modern data centers that are opening all the time to provide the intended functionality. These centers offer a shared pool of resources and want to provide the ideal service by empowering their hardware resources through the addition of excellent software applications, which can be best prepared using OOP principles, the languages that offer reusable code, and the ability to offer constant updates.

OOP languages can prepare software solutions that can save money for all individuals. Modern data centers need solutions for client/server interfaces, client-side applications, and programs that allow for improved division of the available sources. This is all possible by using a modern OOP platform such as Java. It will continue to get better with a more modernized version already in the pipeline, which helps to provide further support towards the creation of cloud applications.

It is also possible to use languages such as Java with complex platforms and other working elements to create effective situations where it is possible to come up with the ideal results. OOP languages are becoming more powerful, as their discrepancies are reduced by modern computational resources. With the possibility of endless heap size available to the programs, you can produce complex inheritance and create programs that take the full advantage of OOP principles in all facets of the application.

12.9 | Understanding the World

The OOP programming is engaging since it allows developers to equate development ideas to the real-world scenarios and find comparative solutions. There are scenarios like "The Diamond Problem" which are difficult to produce in the OOP environment. However, they represent exceptional cases, where some languages like JavaScript offer a way out for programmers. The Diamond Problem, also known as "Deadly Diamond of Death", is a multiple inheritance problem. This problem occurs when two classes, say, Movable and Recliner inherit from Furniture, and class Chair inherits from both Movable and Recliner. Now say, equal method is called for Chair and there is no such method in the Chair class but it is in both Movable and Recliner classes. In this case, upon calling equal on Chair, which equal method should get called as Chair class inherits equal method from both Movable and Recliner? This is called "The Diamond Problem".

Procedural and functional programming paradigms are also helpful. You need to understand when to use OOP to achieve the ideal performance. Using an object-oriented language allows a project manager to divide the development tasks and create sizable chunks of work, where everyone can pitch in without affecting the overall workflow.

A key problem associated with current solutions is that class hierarchies and code supporting elements can often require a large heap size for efficient solutions. With increase in the available resources, it is now easier to implement complex programs that make full use of the important OOP paradigms as independent objects and fully extendable basic classes.

Always remember that we should not force each problem to be resolved using OOP principles. Rather, the ideal way is to start your project management by describing the problem and then work out the possible logical solutions. This will allow you to come up with the relevant OOP ideas in most cases, where you may also learn that some programs may work better using procedural languages.

Summary

OOP is one of several programming paradigms that are in use today. It started out as a programming discipline for creating mainframe computer programs and applications, and became popular gradually. The OOP structure is defined through the principles of encapsulation, abstraction, inheritance, and polymorphism. These principles are incorporated using different tools by various OOP programming languages. Java employs the system of classes and objects to apply these principles.

We have described each of these principles in depth in this chapter. We have presented examples and situations where the OOP paradigm is applied to resolve problems. Another thing that programmers must remember is that each OOP language has its own particular flavor. Some languages cannot be defined as purely object-oriented because they offer other avenues

of creating hierarchy and carrying out a task without creating a strong boundary wall between code implementation and its external behavior.

We also describe the presence of some OOP tools that Java provides. It cannot be termed as a purely OOP language, so we also discussed elements such as an interface, which is present in a different scenario from the implementation of a class object. We described how these principles are placed into action by working out realistic examples.

However, you should remember that this chapter serves as a basic introduction to the OOP world. We have provided some excellent pointers here that you can use to improve your understanding of this programming paradigm.

We completed the chapter by providing an overview of what the future holds for OOP practices. We have identified the key scenarios where OOP languages will continue to excel, while also marking the situations where you should understand the shortfalls of this programming practice.

In this chapter, we have learned answers to the main questions related to OOP:

1. What is object-oriented programming?
2. How does object-oriented programming help create better solutions?
3. What are the principles of object-oriented programming?
4. How do inheritance, polymorphism, encapsulation, and abstraction work?
5. What are the benefits of using abstract classes?
6. How can interface be used to add features to a class?

In Chapter 13, we will learn about generics and collections. We will explore various collection APIs like Map, Set, List, Queue, Collection, SortedSet, SortedMap. And then we will see the concrete implementation classes such as HashMap, Hashtable, TreeMap, LinkedHashMap, HashSet, LinkedHashSet, TreeSet, ArrayList, Vector, LinkedList, and PriorityQueue. We will also learn about Stream API for bulk data operation, forEach, forEachRemaining, removeIf, spliterator, `replaceAll()`, `compute()`, `merge()`, etc.

Multiple-Choice Questions

1. Which one of the following specifiers applies only to the constructors?
   - **(a)** Public
   - **(b)** Protected
   - **(c)** Implicit
   - **(d)** Explicit
2. Which of the following is not a part of OOP?
   - **(a)** Multitasking
   - **(b)** Type Checking
   - **(c)** Polymorphism
   - **(d)** Information hiding
3. We use constructors for which of the following?
   - **(a)** Free memory
   - **(b)** Initializing a newly created object
   - **(c)** Building a user interface
   - **(d)** Creating a sub class
4. run-time polymorphism is known as _____.
   - **(a)** Overriding
   - **(b)** Overloading
   - **(c)** Dynamic Binding
   - **(d)** Coupling
5. What does OOPS stand for?
   - **(a)** Object-Oriented Programming System
   - **(b)** Object-Oriented Processing System
   - **(c)** Object-Oriented Programming Structure
   - **(d)** Object-Oriented Personal Structure

Review Questions

1. What is an abstract class?
2. What is interface? How should we use it?
3. Can you use more than one interface on one particular class?
4. Give a real-life example of polymorphism.
5. How does encapsulation help developers?
6. What is inheritance? Give an example.
7. What is the difference between a class and an object?

Exercises

1. Think of a real-life example of abstraction and draw a diagram. Explain the example.
2. Explain in detail the benefits of inheritance. What would have happened if OOP had missed this principle?

3. Write a program using Eclipse IDE that contains interface and abstract classes. Then create classes that implement these interfaces and extend this abstract class.
4. Explain the concept of overloading and overriding with examples.

Project Idea

Traffic is a day-to-day problem for urban areas. There are various types of vehicles that make it difficult for cyclists and pedestrians. Plan out a software that will allow the city to divide into three types of roads – one for vehicles, one for cyclists, and one for pedestrians. The rule is that a type cannot use the road that are not intended for its use. Make a detailed diagram for this problem.

Recommended Readings

1. Brett McLaughlin, Gary Pollice, David West. 2006. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*. O'Reilly Media: Massachusetts
2. Grady Booch, Robert Maksimchuk, Michael Engle, Jim Conallen, Kelli Houston, Ph.D. Young Bobbi. 2007. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional: Massachusetts
3. Lesson – Object-Oriented Programming Concepts: https://docs.oracle.com/javase/tutorial/java/concepts/index.html